A Parallel Framework for Constraint-Based Bayesian Network Learning via Markov Blanket Discovery

Ankit Srivastava, Sriram P. Chockalingam, and Srinivas Aluru, *Fellow, IEEE*

***Abstract*—Bayesian networks (BNs) are a widely used graphical model in machine learning. As learning the structure of BNs is NP-hard, high-performance computing methods are necessary for constructing large-scale networks. In this article, we present a parallel framework to scale BN structure learning algorithms to tens of thousands of variables. Our framework is applicable to learning algorithms that rely on the discovery of Markov blankets (MBs) as an intermediate step. We demonstrate the applicability of our framework by parallelizing three different algorithms: *Grow-Shrink* (*GS*), *Incremental Association MB* (*IAMB*), and *Interleaved IAMB* (*Inter-IAMB*). Our implementations are available as part of an open-source software called *ramBLe*, and are able to construct BNs from real data sets with tens of thousands of variables and thousands of observations in less than a minute on 1024 cores, with a speedup of up to 845X and 82.5% efficiency. Furthermore, we demonstrate using simulated data sets that our proposed parallel framework can scale to BNs of even higher dimensionality. Our implementations were selected for the reproducibility challenge component of the 2021 student cluster competition (SCC'21), which tasked undergraduate teams from around the world with reproducing the results that we obtained using the implementations. We discuss details of the challenge and the results of the experiments conducted by the top teams in the competition. The results of these experiments indicate that our key results are reproducible, despite the use of completely different data sets and experiment infrastructure, and validate the scalability of our implementations.**

***Index Terms*—Bayesian networks, constraint-based learning, parallel machine learning, gene networks, reproducibility.**

I. Introduction

BAYESIAN networks (BNs), an important subclass of probabilistic graphical models, employ directed acyclic graphs (DAGs) to compactly represent exponential-sized joint probability distributions over a set of random variables. Since BNs enable probabilistic reasoning about direct and indirect interactions between the variables of interest, they have been successfully applied in a wide range of applications in the fields of medical diagnosis [1], gene networks [2], [3], fMRI analysis [4], cybersecurity [5], legal reasoning [6], forensic science [7], etc. Furthermore, the recent focus on the need for explainability in the decisions made by machine learning models [8] has led to a push for the use of inherently interpretable models like BNs [9] in hitherto unexplored areas [10].

Given a data set sampled from a joint probability distribution, exact learning of the corresponding BN structure is NP-hard [11] and therefore a wide range of heuristic methods have been developed for this purpose. However, effective heuristic methods are also compute-intensive and can only construct moderate-scale networks sequentially, which has led to the parallelization of BN structure learning becoming one of the major areas of focus in BN research in recent years. Even so, the works on the topic so far are either specific to a particular algorithm or are application-specific. Broadly applicable parallelization strategies that can be used for constructing large-scale networks have remained elusive thus far.

A. Related Work

Algorithms for BN structure learning can be broadly classified into *score-based* and *constraint-based* methods. *Score-based* methods use a Bayesian metric to evaluate the fitness of a structure given the observed data and attempt to find the highest scoring structure out of all the valid structures. *Constraint-based* approaches, on the other hand, perform repeated applications of conditional independence (CI) tests to eliminate edges between pairs of variables whose dependence can be explained by a conditioning set.

Exact *score-based* algorithms with exponential run-time complexity have been proposed to find the optimal structure for small BNs, i.e., BNs with less than 20 variables [12], [13]. Even parallelization of these exact solutions can only construct networks with a maximum of 37 variables [14], [15], [16].

Heuristics developed for learning BNs can be classified as either *global-search* or *local-to-global*. *Global-search* methods traverse the global space of DAGs to identify an optimal structure. Examples of such methods include *score-based* strategies by [17], [18] and *constraint-based* approaches by [19], [20]. *Local-to-global* methods, on the other hand, first discover the local neighborhood of each variable and then combine these local neighborhoods to obtain the global structure. Multiple *local-to-global* approaches have been proposed in both the

Manuscript received 9 October 2022; revised 18 January 2023; accepted 28 January 2023. Date of publication 13 February 2023; date of current version 5 May 2023. This research was supported in part by the National Science Foundation under Grants OAC-1828187 and OAC-1854828. Recommended for acceptance by Special Section SC21 Reproducibility. *(Corresponding author: Ankit Srivastava.)*

Ankit Srivastava and Srinivas Aluru are with the School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: asrivast@gatech.edu; aluru@cc.gatech.edu).

Sriram P. Chockalingam is with the Institute for Data Engineering and Science, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: srirampc@gatech.edu).

Digital Object Identifier 10.1109/TPDS.2023.3244135

score-based [21], [22] and *constraint-based* [23], [24] categories. We refer the reader to [25] for a comprehensive review.

Compared to exact methods, parallelization of heuristic methods has yielded results with much better scalability. Nikolova et al. [26] developed a parallel *score-based* method that can construct a network with 500 variables in 107 seconds using 1,024 cores. Misra et al. [27] developed a similar approach that learns genome-scale gene networks for *Arabidopsis thaliana*, and can construct a 15,216 variable BN in less than 172 seconds using 1.57 million cores of the *Tianhe-2* supercomputer.

Efforts on parallelizing *constraint-based* methods have been primarily focused on *global-search* methods, in particular on the *PC* algorithm [20]. Madsen et al. [28] parallelized the *PC* algorithm with a shared-memory model for which they achieve a maximum speedup of almost 7 using 12 threads for constructing a network with 2,371 variables. Using a similar approach for parallelizing CI tests with a distributed-memory model, they reported a maximum speedup of about 8 using 10 cores for the same data set. There have been multiple efforts to accelerate the order-independent variant of the *PC* algorithm, referred to as the *stable-PC* algorithm [29]. Le et al. [30] proposed the *parallel-PC* algorithm for the purpose and reported a maximum speedup of 12 using 14 cores for learning a network with 2,810 variables. Schmidt et al. [31] and Zarebavani et al. [32] have proposed strategies for accelerating the *stable-PC* algorithm using GPUs. However, all of the proposed parallelization strategies are specific to either the *PC* or the *stable-PC* algorithm and are not applicable to other *constraint-based* methods.

One of the earliest attempts at parallelizing *local-to-global Constraint-based* algorithms was by Nikolova et al. [33], who focused on parallelizing the *MMHC* algorithm [34] and the *PCMB* algorithm [35]. They reported near perfect scaling for learning neighborhoods of 1,000 variables on up to 512 cores. However, as the authors observed, their approach does not scale when the number of variables or the number of observations are increased. This is because their approach assigns all the computations for determining the local neighborhood of a variable to the same processor. Due to the differences in the computation requirements across variable neighborhoods, such a static assignment of variables to processors leads to load imbalance.

Multiple open-source packages for learning BNs have been developed. The most prominent and well-maintained among them include *bnlearn* [36], *Tetrad* [37], and *pcalg* [38]. However, these implementations are either completely sequential (e.g., *pcalg*) or support only limited intra-node level parallelism (e.g., *Tetrad*). Recently, *bnlearn* added support for parallelizing structure-learning algorithms [39], using a parallelization strategy similar to the one used in [33] and therefore suffers from the same drawbacks as discussed earlier.

B. Contributions

In this paper, we present a parallel framework to scale BN structure learning algorithms to tens of thousands of variables. Our framework is applicable to *local-to-global constraint-based* structure learning algorithms that rely on the discovery of Markov blankets (MBs) as an intermediate step. We identify common components of these algorithms and develop parallel algorithms for each of these components. Subsequently, we demonstrate the applicability of our framework by using it to develop parallel versions of three different algorithms: the *Grow-Shrink* (*GS*) algorithm [23], the *Incremental Association MB* (*IAMB*) algorithm [24], and the *Interleaved IAMB* (*Inter-IAMB*) algorithm [24]. We also introduce different algorithmic techniques that improve run-time performance of these algorithms both sequentially and in parallel, such as optimizations for CI testing and load balancing.

We demonstrate the scalability of these algorithms using real data sets to learn genome-scale gene networks for the organisms *Saccharomyces cerevisiae* and *Arabidopsis thaliana* – networks with tens of thousands of variables from thousands of observations. The experiments show that our optimized implementations of the three algorithms achieve significant sequential speedup over the popular *bnlearn* package in learning these networks. Further, our proposed parallel versions of these algorithms are able to learn the networks in less than a minute on 1024 cores, compared to almost 14 hours required by our sequential implementation and close to 24 hours required by *bnlearn*. Using simulated data sets, we show that our algorithms are scalable to learning networks with even larger number of variables and can reduce the time required for the purpose from more than 25 hours sequentially to less than two minutes on 1,024 cores.

Using the proposed framework, our implementations of the three algorithms are able to achieve a speedup of up to 845X corresponding to a strong scaling efficiency of 82.5% on 1024 cores. Even though we demonstrate the utility of our framework in constructing gene networks, the experiments on simulated data sets show that our framework can enable learning of higher-dimensional BNs at scale in other application areas, e.g., fMRI analysis [4], and potentially enable their adoption in other fields where the time required for learning large-scale BNs has heretofore been a deterrent.

II. Background

A BN is a graphical representation of a joint probability distribution of a set of n random variables $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$, denoted by the pair $(\mathcal{G}, P)$, where $\mathcal{G}$ is a DAG of n vertices corresponding to each variable in $\mathcal{X}$ and $P(\mathcal{X})$ is the joint probability distribution that decomposes into conditional probability distributions as follows:

$$P(X_1, \ldots, X_n) = \prod_i P(X_i|\mathcal{R}(X_i)),$$

where $\mathcal{R}(X_i)$ denotes the set of parents of X_i in $\mathcal{G}$. We use upper-case alphabets (e.g., X, X_i, Y) to represent random variables and calligraphic upper-case alphabets (e.g., $\mathcal{X}, \mathcal{R}, \mathcal{S}$) to represent sets of random variables. The values that a random variable can take are represented using lower-case letters (e.g., a, b, c). A BN satisfies the *faithfulness* condition if $\mathcal{G}$ entails all and only the CIs present in $P(\mathcal{X})$. We assume faithfulness in this paper and refer the reader to [40] for details on faithfulness and entailed CIs. Fig. 1 shows an example BN for the six variables

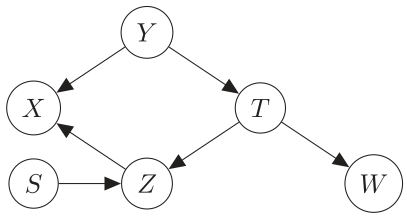

Fig. 1. An example BN for a set of six random variables, $\mathcal{X} = \{S, T, W, X, Y, Z\}$. The probability distribution $P(\mathcal{X})$ decomposes as $P(S)P(Y)P(T|Y)P(W|T)P(Z|\{S,T\})P(X|\{Y,Z\})$.

$\{S, W, T, X, Y, Z\}$. The directed arrows in the BN represent parent-child relationships, e.g., $\mathcal{R}(T)$ is $\{Y\}$ and T is present in both $\mathcal{R}(W)$ and $\mathcal{R}(Z)$ in the BN shown in Fig. 1.

We represent CI between two random variables X and Y given a third variable Z as $I(X, Y|Z)$ and conditional dependence as $\neg I(X, Y|Z)$. Given a set of observations, tests for conditional independence can be conducted using different statistical tests for discrete and continuous variables. For discrete variables, the most common method of determining $I(X, Y|Z)$ is by computing the G^2 statistic [41] as follows:

$$G^2 = 2 \sum_{c \in Z} \sum_{a \in X} \sum_{b \in Y} s_{abc} \ln \frac{s_{abc} s_c}{s_{ac} s_{bc}}, \qquad (1)$$

where s_c is the number of observations in which $Y = c$, s_{ac} is the number of observations in which $X = a$ and $Z = c$, s_{abc} is the number of observations in which $X = a$, $Y = b$, and $Z = c$, etc. Under the null hypothesis that the CI holds, the G^2 statistic is asymptotically distributed as chi-squared with the degrees of freedom computed as $(r_X - 1) \cdot (r_Y - 1) \cdot r_Z$ where r_X is the arity of the variable X, i.e., the number of different values that X can take, etc. The *p-value* of the G^2 test is computed as the probability that the G^2 statistic was drawn from the chi-squared distribution. If the *p-value* is lower than a significance threshold, denoted by α, the null hypothesis is rejected and $\neg I(X, Y|Z)$ is determined to be true. Lower *p-value* indicates stronger dependence and therefore we use the additive inverse of *p-value* for quantifying the strength of association between the variables, represented by $Assoc(X, Y|Z)$. We also use the $I(\cdot, \cdot|\cdot)$, $\neg I(\cdot, \cdot|\cdot)$, and $Assoc(\cdot, \cdot|\cdot)$ notations for sets of variables.

The set of parents and children of a variable T in a BN, represented by $\mathcal{PC}(T)$, consists of variables that are dependent on T given any conditioning set not containing the two variables, i.e., $X \in \mathcal{PC}(T)$ if and only if $\neg I(X, T|\mathcal{S}) \forall \mathcal{S} \subseteq \mathcal{X} \setminus \{X, T\}$. For example, in the BN shown in Fig. 1, $\mathcal{PC}(T)$ is $\{Y, Z, W\}$. The MB of a variable T is defined as a set of variables, denoted by $\mathcal{MB}(T)$, that renders T independent of other variables, i.e., $I(X, T|\mathcal{MB}(T)) \forall X \in \mathcal{X} \setminus (\mathcal{MB}(T) \cup \{T\})$. Assuming faithfulness, $\mathcal{MB}(T)$ is made up of the variables in $\mathcal{PC}(T)$ and the parents of the children of T. In the example BN from Fig. 1, $\mathcal{MB}(T)$ is $\mathcal{PC}(T) \cup \{S\} = \{Y, Z, W, S\}$. Note that under the faithfulness assumption, $\mathcal{MB}$ implies a symmetric relation similar to $\mathcal{PC}$, i.e., if $X \in \mathcal{MB}(T)$ then $T \in \mathcal{MB}(X)$.

In this paper, we focus on the *local-to-global constraint-based* methods for constructing BNs, which learn the $\mathcal{PC}$ set for every variable separately and then combine them to get the complete network. These algorithms belong to one of the following two subclasses: *blanket learning* or *direct learning*. *Blanket learning* algorithms identify the $\mathcal{MB}$ for each variable as a first step and then identify the subset of variables in the $\mathcal{MB}$ which also belong to the $\mathcal{PC}$, e.g., *GS*, *IAMB*, *Inter-IAMB*, etc. *Direct learning* algorithms, on the other hand, construct the $\mathcal{PC}$ set for every variable without any intermediate steps, e.g., *MMPC* [34], *HITON-PC* [42], etc. In both types of learning, the skeleton of the BN (i.e., an undirected representation of the DAG) is constructed first, followed by the orientation of the edges. Since the skeleton constructed for real data sets is very sparse and the edge orientation algorithm is linear in the number of edges, a miniscule portion of the total run-time (less than 0.01%) is spent in directing the edges. Accordingly, we do not focus on developing a parallel method for this step.

III. Parallel Algorithm

We propose a parallel framework which enables users to develop and implement an efficient parallel version of any *blanket learning* strategy for constructing BNs. First, we introduce the notations used and state the key assumptions in Section III-A. In Section III-B, we describe the sequential version of *blanket discovery* algorithms. Then, we discuss the components of our proposed parallel framework in Section III-C. Finally, using these components we present parallel versions of three *blanket learning* algorithms – *GS*, *IAMB*, and *Inter-IAMB* in Section III-D.

A. Assumptions and Notations

Similar to the other $\mathcal{MB}$ construction algorithms, we assume an ordering of the input variables in $\mathcal{X}$, i.e., $X_1 < X_2 < \ldots < X_n$. We also assume, similar to other parallel algorithms, that the input dataset D with m observations for n variables is available locally on all the processors. For the computations of the run-time complexity, we make the standard assumption that conducting CI tests and computing $Assoc(\cdot)$ values takes constant time. In order to model the communication time requirements of the proposed parallel algorithms, we assume a parallel distributed system with p processors that requires τ time units to setup communication between processors and μ time units per word to send a message from one processor to another.

In our framework, the key data structure that we use is a list of tuples, referred to as *c-scores*. Elements of *c-scores* are of the form $\langle X, Y, \theta_{XY} \rangle$, where X and Y are variables and θ_{XY} is a numeric value. At any point during the execution of the algorithms, if $\langle X, Y, \theta_{XY} \rangle$ is an element in *c-scores*, then the variable Y is a potential candidate for addition to the MB set of the target variable X, i.e., $\mathcal{MB}(X)$. The third element, θ_{XY}, is the score for adding Y to $\mathcal{MB}(X)$ and is used to select the best candidate for every target variable. For the algorithms presented in this paper, we use the associativity of a target X and the candidate Y given the current MB of the X as the score, i.e.,

$\theta_{XY} = Assoc(X, Y|\mathcal{MB}(X))$. Apart from the *c-scores* list, we also maintain a list, denoted as $variables$, that contains all the variables for which the MB sets are to be computed.

In order to construct the BN skeleton, *blanket learning* algorithms need to identify the MB sets for all the variables. Accordingly, we initialize the $variables$ list with all the variables in $\mathcal{X}$. Since MB discovery generally starts with empty MB sets, $\mathcal{MB}(T)$ is initialized to $\emptyset\ \forall T \in variables$. We initialize the *c-scores* list with a tuple each for all the possible candidates for all the variables in $\mathcal{X}$ and set all the scores to zero, i.e., the list is initialized with elements from the set $\{\langle X, Y, 0\rangle | X \in \mathcal{X}, Y \in \mathcal{X} \setminus (\{X\} \cup \mathcal{MB}(X))\}$. At the beginning of the algorithm, there is a tuple in *c-scores* corresponding to each of the $n^2 - n$ ordered variable pairs. Furthermore, the tuples in the *c-scores* list are initialized in the ascending order of the first variable and then of the second variable. Therefore, all the tuples with the candidate variables corresponding to the same target variable are arranged in a contiguous manner in the list.

When executing the algorithms on p processors, the *c-scores* list is initialized in a similar fashion but is block distributed among all the processors. The corresponding list on the processor j is denoted by *c-scores*$_j$ and its size is bounded by $\lceil \frac{n^2-n}{p} \rceil$. The list $variables_j$ is initialized with all the variables for which the processor j computes the MB sets, i.e., it includes all the elements from the set $\{X | \langle X, Y, \theta_{XY}\rangle \in$ *c-scores*$_j\}$. Since the *c-scores* list is ordered such that tuples with the same first variable are contiguous, the size of $variables_j$ is bounded by $O(\frac{n}{p})$. In the distributed setting, $\mathcal{MB}(T)$ is initialized on every processor for all $T \in variables_j$. Note that, for two different processors i and j and some variable T, both $variables_i$ and $variables_j$ may contain T. In such cases, both processors i and j compute $\mathcal{MB}(T)$.

B. Algorithm

Prior to describing the components of the proposed parallel framework, we briefly outline how the algorithms of interest proceed on a single processor. BN learning via MB discovery, in general, is comprised of four phases – *Grow*, *Shrink*, *Symmetry Correction*, and *Construct $\mathcal{PC}$ from $\mathcal{MB}$*. During the *Grow* phase, the MB for a variable T is grown by adding a variable to $\mathcal{MB}(T)$ from among the available candidates. In the *Shrink* phase, one or more variables are removed from $\mathcal{MB}(T)$ if they are independent of T given the other variables in the current MB set. After identifying the candidate MB sets for all the variables in one or more *Grow* and *Shrink* iterations, *Symmetry Correction* is performed to obtain symmetrically consistent MB sets. Finally, for learning the skeleton of BN using the MBs, the algorithms *Construct $\mathcal{PC}$ from $\mathcal{MB}$* for every variable, i.e., a variable X in $\mathcal{MB}(T)$ is included in $\mathcal{PC}(T)$ if no subset of $\mathcal{MB}(T)$, with the exclusion of X (or alternatively $\mathcal{MB}(X)$ with the exclusion of T) can render X and T conditionally independent.

We now describe in detail the *GS*, *IAMB*, and *Inter-IAMB* algorithms in terms of the *c-scores* and $variables$ lists, defined and initialized as per Section III-A. In a *Grow* phase iteration, scores are first updated for all the tuples in the current *c-scores* list. This is accomplished by updating θ_{XY} with $Assoc(T, Y|\mathcal{MB}(X))$ for all $\langle T, Y, \theta_{TY}\rangle \in$ *c-scores*. Then, using the updated scores, a candidate is selected for every variable. More specifically, the *Grow* phase in the *IAMB* and the *Inter-IAMB* algorithms picks the candidate with the maximum score, i.e., the tuple $\langle T, Z, \theta_{TZ}\rangle$ is picked for T if

$$\langle T, Z, \theta_{TZ}\rangle = \underset{\langle T, Y, \theta_{TY}\rangle \in c\text{-}scores}{\arg\max}\ \theta_{TY}. \quad (2)$$

The *GS* algorithm, on the other hand, picks for a variable T the first candidate that shows dependency with T. As mentioned in Section II, we use the additive inverse of *p-value* of the G^2 test $I(T, X|\mathcal{MB}(T))$ as $Assoc(T, Y|\mathcal{MB}(T))$. Therefore, candidate selection for *GS* can be accomplished by identifying the first tuple with a score greater than the additive inverse of the significance threshold ($-\alpha$), i.e., a tuple $\langle T, Z, \theta_{TZ}\rangle$ is selected for T if

$$\langle T, Z, \theta_{TZ}\rangle \text{ is first entry in } c\text{-}scores \text{ s.t. } \theta_{TY} \geq -\alpha. \quad (3)$$

In both the cases, if such a tuple is found, then Z is added to $\mathcal{MB}(T)$ and $\langle T, Z, \theta_{TZ}\rangle$ is removed from the *c-scores* list.

During the *Shrink* phase, we examine all the $\mathcal{MB}$ sets and remove a variable X from $\mathcal{MB}(T)$ if $I(T, X|\mathcal{MB}(T) \setminus \{X\})$ holds. *Blanket learning* algorithms differ on how *Grow* and *Shrink* phases are iterated. Both *GS* and *IAMB* execute multiple iterations of *Grow* phase followed by a single *Shrink* phase, whereas the *Inter-IAMB* algorithm alternates between *Grow* and *Shrink* phases until all the $\mathcal{MB}$ sets stop changing.

After the one or more iterations of *Grow* and *Shrink* phases, MB construction proceeds to *Symmetry Correction*, in which we verify whether $T \in \mathcal{MB}(Y) \iff Y \in \mathcal{MB}(T)$ and when this assertion fails for a pair (T, Y), we remove the offending variables from the respective $\mathcal{MB}$ sets. Finally, the $\mathcal{PC}$ sets are learned by verifying CI for every subset of $\mathcal{MB}$, after which the DAG is obtained by orienting the undirected edges added from the $\mathcal{PC}$ sets.

C. Parallel Framework Components

We now discuss the key components of our proposed framework – parallel algorithms for all the four steps described in Section III-B. We designed these components using common parallel primitives such as `all-reduce`, `scan`, shift permutations, and parallel sorting.

1) Grow Phase: Our parallel algorithm for *Grow* phase is based on the following two key insights: (i) The MB sets for all the variables are required for constructing the skeleton. Further, for addition to the MB set of a variable, all the other variable are considered a candidate. Therefore, we consider all the variable pairs in parallel, using the distributed *c-scores* list. (ii) The time taken in conducting a CI test (or computing $Assoc(\cdot)$) is proportional to the size of the conditioning set. Therefore, we designed this component such that the CI tests (and $Assoc(\cdot)$ computations) with the same conditioning set sizes are conducted in parallel.

The pseudo-code for our parallel *Grow* phase is shown in Algorithm 1. As discussed in Section III-B, the MB construction algorithms use different heuristics to select the next variable

Algorithm 1: Parallel *Grow* Phase.

```
function GROW-PHASE():
    Input: D, $c\text{-}scores$, $variables$, current $\mathcal{MB}(\cdot)$ sets,
        APPLY-HEURISTIC, REDUCE-HEURISTIC
    Output: Updated $\mathcal{MB}(\cdot)$ sets
    parallel $j$ = processor's rank do
        for $\langle T, Y\rangle \in$ c-pairs$_j$ do
            $\theta_{TY} \leftarrow Assoc(T, Y|\mathcal{MB}(T), \mathsf{D})$
            Add/Update $\langle T, Y, \theta_{TY}\rangle$ to $c\text{-}scores_j$
        $g\text{-}select_j(T) \leftarrow$ nil, $\forall T \in variables_j$
        for $T \in variables_j$ do
            $ts \leftarrow \langle T, X, \theta_{TX}\rangle \in c\text{-}scores_j, \forall X \in \mathcal{X}$
            $g\text{-}select_j(T) \leftarrow$ APPLY-HEURISTIC ($ts$)
        REDUCE-HEURISTIC ($c\text{-}scores$, $g\text{-}select$)
        for $T \in variables_j$ do
            $Z \leftarrow g\text{-}select_j(T)$
            if $\neg I(T, Z|\mathcal{MB}(T), \mathsf{D})$ then
                $\mathcal{MB}(T) \leftarrow \mathcal{MB}(T) \cup \{Z\}$
                Remove $\langle T, Z, \theta_{TZ}\rangle$ from $c\text{-}scores_j$
```

to be added to the current MB set. In order to accommodate these differences, our proposed *Grow* algorithmic component requires two functions as arguments: APPLY-HEURISTIC and REDUCE-HEURISTIC. The function APPLY-HEURISTIC accepts a slice of the *c-scores* list corresponding to a variable T such that it contains $\langle T, X, \theta_{TX}\rangle$ for all the candidates X, and returns the candidate most suitable for addition to $\mathcal{MB}(T)$. For example, the APPLY-HEURISTIC selects a candidate as per (2) for the *IAMB* and the *Inter-IAMB* algorithms and as per (3) for the *GS* algorithm. The REDUCE-HEURISTIC function accumulates the variable selection results from all the processors to identify for each variable T, the best candidate to be added to its $\mathcal{MB}(T)$.

In Algorithm 1, the local *c-scores* list is updated with the computed *Assoc* values first (lines 3–5), which takes $O(\frac{n^2}{p})$ time (assuming that *Assoc* computations take constant time). The selection heuristic is then applied for each variable (lines 7–9) followed by the accumulation of results across processors (line 10). The run-time of these operations depends on the heuristic used by the specific algorithm. For most MB algorithms, including *GS*, *IAMB*, and *Inter-IAMB*, two segmented parallel `scan` operations are sufficient for accumulating the results from all the processors because the underlying operators are associative. Note that these parallel `scan` operations exploit the contiguous presence of all the tuples $\langle T, Y, \theta_{TY}\rangle$ corresponding to a target variable T in *c-scores*. Therefore, APPLY-HEURISTICand REDUCE-HEURISTICtake $O(\frac{n^2}{p} + \log p)$ time for computation and $O((\tau + \mu)\log p)$ time for communication. The $\mathcal{MB}$ sets updated are local to the processor and therefore the number of $\mathcal{MB}$ sets updated in a processor is bounded by $O(\frac{n}{p})$. Finally, we add the selected variables to the $\mathcal{MB}$ sets and update *c-scores* (lines 11–15), bounded by $O(\frac{n}{p})$. Therefore, the *Grow* phase algorithmic component takes $O(\frac{n^2}{p} + \log p)$ time for growing all the $\mathcal{MB}$ sets by one variable. The only collective

Algorithm 2: Parallel *Shrink* Phase.

```
function SHRINK-PHASE():
    Input: D, $variables$, current $\mathcal{MB}(\cdot)$ sets
    Output: Updated $\mathcal{MB}(\cdot)$ sets
    parallel $j$ = processor's rank do
        for $T \in variables_j$ do
            for $Z \in \mathcal{MB}(T)$ do
                if $I(T, Z|\mathcal{MB}(T) \setminus \{Z\}, \mathsf{D})$ then
                    $\mathcal{MB}(T) \leftarrow \mathcal{MB}(T) \setminus \{Z\}$
```

Algorithm 3: Parallel *Symmetry Correction*.

```
function SYMMETRY-CORRECTION():
    Input: $variables$, asymmetric $\mathcal{MB}(\cdot)$ sets
    Output: Symmetry corrected $\mathcal{MB}(\cdot)$ sets
    parallel $j$ = processor's rank do
        $sc\text{-}pairs_j \leftarrow$ empty list of variable pairs
        for $X \in variables_j$ do
            if $j = 0$ or $X \notin variables_{j-1}$ then
                for $Y \in \mathcal{MB}(X)$ do
                    if $X < Y$ then
                        Insert $\langle X, Y\rangle$ into $sc\text{-}pairs_j$
                    else
                        Insert $\langle Y, X\rangle$ into $sc\text{-}pairs_j$
        Parallel sort $sc\text{-}pairs$, first by $X$ then by $Y$
        Remove all unique $\langle X, Y\rangle$ from $sc\text{-}pairs$
        Reset all $\mathcal{MB}(\cdot)$ sets to $\emptyset$
        for $\langle X, Y\rangle \in sc\text{-}pairs_j$ do
            $\mathcal{MB}(X) \leftarrow \mathcal{MB}(X) \cup \{Y\}$
            $\mathcal{MB}(Y) \leftarrow \mathcal{MB}(Y) \cup \{X\}$
```

communication in this component is for reducing the heuristic computations across the processors, which takes $O((\tau + \mu)\log p)$ time.

2) Shrink Phase: Our proposed parallel component for *Shrink* phase is shown in Algorithm 2. Here, the only task is to remove those variables in $\mathcal{MB}$ which are independent given the rest of the $\mathcal{MB}$, which is accomplished by a loop over all the $\mathcal{MB}$ sets (lines 3–6). The run-time for the parallel *Shrink* phase is proportional to the size of the $\mathcal{MB}$ sets for all the variables on the processor, which is bounded by $n \times O(\frac{n}{p}) = O(\frac{n^2}{p})$. This component requires no communications.

3) Symmetry Correction: The proposed parallel component for checking the symmetry of the $\mathcal{MB}$ sets, shown in Algorithm 3, is based on the method developed by [33]. It proceeds by creating *sc-pairs*, a list of ordered tuples for every member of $\mathcal{MB}$ set (lines 3–10) followed by parallel sorting to identify the asymmetric $\mathcal{MB}$ members (lines 11–12). The $\mathcal{MB}$ sets are then updated to reflect the symmetry correction (lines 13–16). The time to construct *sc-pairs*, remove unique tuples, and update $\mathcal{MB}$ sets is bounded by $O(\frac{n^2}{p})$. Parallel sorting can be accomplished by any comparison based sort such as parallel bitonic sort, which takes $O(\frac{n^2}{p}\log\frac{n^2}{p} + \frac{n^2}{p}\log^2 p)$ and

Algorithm 4: Parallel *Construct $\mathcal{PC}$ from $\mathcal{MB}$*.

```
1  function GET-PC():
       Input: D, $variables$, complete $\mathcal{MB}$ sets, $neighbors$
       Output: Updated $neighbors$ representing $\mathcal{PC}$ sets
2      parallel $j$ = processor's rank do
3          for $X \in variables_j$ do
4              for $Y \in \mathcal{MB}(X)$ do
5                  if $|\mathcal{MB}(X)| < |\mathcal{MB}(Y)|$ then
6                      $\mathcal{B} \leftarrow \mathcal{MB}(X) \setminus \{Y\}$
7                  else
8                      $\mathcal{B} \leftarrow \mathcal{MB}(Y) \setminus \{X\}$
9                  if $\neg I(X, Y|\mathcal{S}, D) \forall \mathcal{S} \subseteq \mathcal{B}$ then
10                     Insert $\langle X, Y\rangle$ into $neighbors_j$
```

$O(\tau \log^2 p + \mu \frac{n^2}{p} \log^2 p)$ time for computation and communication respectively. Collective communication is also required during the removal of the unique tuples to identify tuple pairs that cross processor boundary. This is accomplished by a pair of shift permutations that take $O(\tau + \mu)$ time. To summarize, the run-time of this component is dominated by the run-time of parallel sort which takes $O(\frac{n^2}{p} \log \frac{n^2}{p} + \frac{n^2}{p} \log^2 p)$ computation time and $O(\tau \log^2 p + \mu \frac{n^2}{p} \log^2 p)$ communication time.

4) Construct $\mathcal{PC}$ From $\mathcal{MB}$: Our parallel algorithm to construct the skeleton of the BN from the $\mathcal{MB}$ sets is shown in Algorithm 4. This component tries to identify a conditioning set for each element Y in $\mathcal{MB}(X)$, that can render X conditionally independent from Y (lines 3–10). If no such conditioning set can be identified for the pair X, Y, the algorithm inserts $\langle X, Y\rangle$ into $neighbors$, a distributed list of tuples (line 10). Note that this component requires the $\mathcal{MB}$ sets of both X and Y and therefore the complete $\mathcal{MB}$ sets should be made available on all the processors before GET-PC is called. In the worst case, this component has a run-time complexity of $O(\frac{n^2}{p} 2^r)$, where $r = \max_{X\in\mathcal{X}} |\mathcal{MB}(X)|$. Since the maximum value of r is low for real networks (less than 6 in our experiments), the corresponding exponential term can be considered constant. Hence, this phase takes the least significant time of all the four phases in practice (less than 0.1%). This component requires no collective communications.

D. Parallel BN Construction

Using the four algorithmic components discussed in Section III-C, many *blanket learning* algorithms can be implemented. Here, we present efficient parallel versions of the following three algorithms: *GS*, *IAMB*, and *Inter-IAMB*. The *GS* algorithm as well as the *IAMB* algorithm can be implemented using the parallel skeleton construction algorithm presented in Algorithm 5. As discussed in Section III-B, the only distinction between the *GS* algorithm and the *IAMB* algorithm is how the next variable is selected in the *Grow* phase and this difference can be abstracted using the APPLY-HEURISTICand REDUCE-HEURISTICfunctions. Given these algorithm-specific functions (defined as per (2) and (3)), the proposed parallel versions

Algorithm 5: Parallel Skeleton Algorithm.

```
1  function CONSTRUCT-SKELETON-GSIAMB():
       Input: D, APPLY-HEURISTIC,
              REDUCE-HEURISTIC
       Output: $\mathcal{PC}(T)$ sets for all $T \in \mathcal{X}$
2      parallel $j$ = processor's rank do
3          Initialize $c\text{-}scores_j$, $variables_j$, $\mathcal{MB}(\cdot)$ as
             described in Section 3.1
4          Initialize $neighbors$ as empty list of tuples
5          repeat
6              GROW-PHASE(D, $c\text{-}scores$, $variables$,
                 $\mathcal{MB}$, APPLY-HEURISTIC,
                 REDUCE-HEURISTIC)
7          until no $\mathcal{MB}$ changes on any of the processors
8          SHRINK-PHASE(D, $variables$, $\mathcal{MB}$)
9          SYMMETRY-CORRECTION($variables$, $\mathcal{MB}$)
10         Synchronize $\mathcal{MB}(\cdot)$ across all the processors
11         GET-PC(D, $variables$, $\mathcal{MB}$, $neighbors$)
```

Algorithm 6: *Grow* - *Shrink* Loop for *Inter-IAMB*.

```
5  repeat // Grow-Shrink loop
6      GROW-PHASE(D, $c\text{-}scores$, $variables$, $\mathcal{MB}$,
         APPLY-HEURISTIC, REDUCE-HEURISTIC)
7      + SHRINK-PHASE(D, $variables$, $\mathcal{MB}$)
8  until no $\mathcal{MB}(X)$ changes on any of the processors
9  - SHRINK-PHASE(D, $variables$, $\mathcal{MB}$)
```

of both these algorithms proceed the same way. The requisite distributed lists and variables are initialized first (lines 3–4), following which these algorithms execute the GROW-PHASEin a loop until convergence (lines 5–7). After SHRINK-PHASEand SYMMETRY-CORRECTION(lines 8–9), there is a synchronization step for collecting the $\mathcal{MB}(\cdot)$ for all the variables on all the processors (line 10). Finally, GET-PCis called to construct the skeleton for the BN in parallel (line 11).

Summing up the run-times of the four components, the computational run-time complexity of Algorithm 5 is

$$O\left(\frac{n^2}{p}\left(\log^2 p + \log\frac{n}{p} + k + 2^r\right) + k \log p\right).$$

where $r = \max_{X\in\mathcal{X}} |\mathcal{MB}(X)|$ and k is the number of times the algorithm executes the *Grow* component.

Apart from the communication costs incurred by the four components, Algorithm 5 also requires collective communications for (i) identifying if any of the $\mathcal{MB}$ sets changed during a *Grow* iteration, and (ii) synchronization of the $\mathcal{MB}$ sets. Using a bit set representation of the $\mathcal{MB}$ sets, both of these operations can be performed using `all-reduce`, which takes $O((\tau + \mu \log\log n) \log p)$ time. Hence, the communication run-time of this algorithm is

$$O\left(\tau\left(\log^2 p + k \log p\right) + \mu\left(\frac{n^2}{p} \log^2 p + k \log p \log\log n\right)\right).$$

The parallel version of the *Inter-IAMB* BN skeleton construction requires only a minor change from Algorithm 5. The modifications, highlighted in Algorithm 6, are the introduction of the

Shrink phase in the $\mathcal{MB}$ update loop (line 7) and the removal of it from outside the loop (line 9). Since the *Shrink* phase takes $O(\frac{n^2}{p})$ time with no communication, the time complexity of the modified algorithm remains the same as that of Algorithm 5.

IV. Implementation

We implemented our framework using *C++* and *MPI* conforming to the *C++14* and *MPI 3.1* standards, respectively. Our implementations are available as part of an open-source software called *ramBLe* [43].

A. Sequential Implementation

Bnlearn [36] is a popular *R* package that supports a wide range of *score-based* and *constraint-based* algorithms for learning BNs, including the three algorithms that we focus on. The package has been used in multiple recent studies for the construction and analysis of BNs [44], [45], [46], [47]. Even though the top-level logic for most of the algorithms supported by *bnlearn* is implemented using *R*, the computationally intensive tasks such as the computations for conducting the CI tests are implemented in *C*. Hence, in spite of interfacing with an interpreted language, *bnlearn* is able to achieve performance comparable to that of a compiled language.

Our implementations differ from *bnlearn*'s implementations because of the ambiguity in the specification of the *GS* algorithm and the choice of internal data structures. For efficiency purposes, we used different data structures than the ones used by *bnlearn* for some of the underlying tasks. For example, *bnlearn* uses arrays for storing the indices of the variables in a set. But, we use bit sets which enables us to use SIMD instructions for some set operations and also reduce the message sizes during communication. This modification, however, may alter the order in which the variables are considered by the algorithms in some cases. Since CI testing using real data sets is imperfect and any errors in the CI tests may change the behavior of the *constraint-based* algorithms, the BNs learned by such algorithms are known to be sensitive to the ordering of the variables [29], [41], [48]. In order to ensure that our choices for efficiency do not affect the accuracy of the learned network, we validate our implementations against *bnlearn* in Section V-B. Our experiments show that these choices help us achieve considerable speedup over *bnlearn* without significantly impacting the learned network structure.

B. Statistic Computation Strategies

In our earlier discussion on the proposed parallel BN algorithms, we used the standard assumption that CI tests can be conducted in $O(1)$ time. However, prior studies have estimated that more than 90% of the time in *constraint-based* learning is spent in aggregating counts from observation data for the CI tests [49]. Correspondingly, we observed that computing the G^2 statistic took between 94% and 99% of the total run-time for learning the network sequentially in our experiments in Section V. Therefore, both efficiently conducting the CI tests as well as reducing the number of CI tests are essential for good run-time performance of the learning algorithms in practice.

1) Counting Strategies: In order to conduct the CI test $I(X, Y|Z)$, using the G^2 statistic (1), the count of the number of observations $s_{abc}, s_{ac}, s_{bc},$ and s_c corresponding to each combination of $X = a$, $Y = b$, and $Z = c$ is required. In BN structure learning implementations, two types of approaches have been used to compute these counts. The most common approach, also used by *bnlearn*, is to compute the counts when they are required, either by scanning the complete data set to fill up contingency tables or using advanced strategies based on bit maps or radix sort [27], [49]. An alternate approach is to pre-process the data set and create an index data structure, e.g., a hash table or an *ADtree* [50], which can be used to retrieve the counts in almost constant time during learning. As discussed by [49], the latter category of approaches require significant pre-processing time which can not be amortized by the corresponding gains during the learning of sparse networks. Thus, we focused on the approaches in the former category and implemented the contingency table based approach as well as the two other strategies from the *SABNAtk* library [49]. We observed that the contingency table based approach outperformed the other two approaches for the data sets that we experimented with. Consequently, we report the run-times using the contingency table based approach in Section V. Nevertheless, our framework can be easily extended to use other counting strategies.

2) Optimizing the GS Algorithm: We reduced the number of $Assoc(\cdot)$ computations in our implementation of the *GS* algorithm based on the observation that, in each iteration of the *Grow* phase, the update of the $c\text{-}scores$ list for a target variable X can be terminated as soon as the first score θ_{XY} which is greater than or equal to $-\alpha$ is computed. This is because the corresponding candidate Y will be the one picked by the algorithm for addition to $\mathcal{MB}(X)$ in that iteration, as per (3). Using this optimization, we are able to reduce the sequential run-time of our implementation of the *GS* algorithm, as shown in Section V-B. Notice that this optimization is useful even in a parallel implementation, when the $c\text{-}scores$ list corresponding to a target X may be distributed across multiple processors. However, since the score updates happen concurrently on all the processors, a processor j will stop the updates for X only after finding the first viable candidate for X in its local list $c\text{-}scores_j$. Therefore, if a suitable candidate for X exists on a processor $i < j$, then extra work is done on the processor j as compared to the sequential execution. We discuss the effect of this optimization on the scaling performance of the *GS* algorithm in Section V-D.

C. Load Balancing

Construction of a BN in parallel starts with a block distribution of the list of candidate tuples, $c\text{-}scores$, to all the processors. In every iteration of the *Grow* phase, one tuple is selected for every variable and removed from the $c\text{-}scores$ list. Furthermore, if the MB of a variable stops changing, then all the candidate tuples corresponding to that variable are removed from the list as well. After a few iterations, these removals can lead to a disparity between the size of the *c-scores* list across the processors. Since the time taken by a processor in an iteration of the *Grow* phase is proportional to the size of the *c-scores* list on that processor, the run-time of an iteration is determined by the the processor with the maximum number of tuples. This load imbalance between processors can, therefore, increase the total time required for learning the $\mathcal{MB}$ sets, which accounts for more than 99% of the total run-time for learning the network.

We mitigate the load imbalance problem by a stable block redistribution of the remaining candidate tuples at the end of an iteration. Specifically, we use an `MPI_Alltoallv` call to redistribute the remaining elements of the *c-scores* list so that it is block distributed while maintaining the original order of the tuples. However, since redistribution is expensive and adds to the total run-time, we redistribute only if the imbalance is severe. For determining the severity, we compute the imbalance at the end of every iteration as the ratio of the maximum size of the list on any processor to the minimum size of the list on any processor, i.e.,

$$\text{Imbalance} = \frac{\max_{0\leq j<p} |c\text{-}scores_j|}{\min_{0\leq j<p} |c\text{-}scores_j|}. \tag{4}$$

We use this metric because both the maximum as well as the minimum load are vital in quantifying the load imbalance and the higher the difference between the two the higher the imbalance. In our implementation, redistribution is done if the computed imbalance is greater than a user-specified threshold. We observed that setting this threshold to 2.0 resulted in optimal performance for every combination of data sets and number of processors in our experiments in Section V. Therefore, we use it as the default value for the threshold in our framework. We study the load imbalance and its effect on the total run-time further in Section V-C.

V. Experiments and Results

We performed our experiments on Georgia Tech's 484-node Hive cluster, where each node has a 2.7 GHz 24-core Intel Xeon 6226 processor and a minimum of 192 GB of main memory. The nodes run *RHEL 7.6* operating system and are connected via EDR (100 Gbps) InfiniBand. For the scalability experiments, we used a maximum of 64 nodes on this cluster. We compiled the source code, implemented with *C++14* and *MPI*, using *gcc v9.2.0* with `-O3 -march=native` optimization flags and *MVAPICH2 v2.3.3* implementation of MPI. We report the run-times measured by assigning 16 MPI processes per node and averaging the run-times over 5 different runs. Similar scaling pattern is observed when all the cores of a node, i.e., 24 processes per node, are used.

In our experiments, we observed that the first calls to the MPI `all-to-all` collectives took significantly longer time than the subsequent calls. Therefore, we warm up both `MPI_Alltoall` and `MPI_Alltoallv` by calling them with one byte on each processor. The time taken by the warm-up phase increases from 0.6 seconds for 32 processes to 5.8 seconds for 512 processes and is not included in the reported run-times. The warm-up takes negligible time when all the processes are within a node and, also, with 1024 processes.

TABLE I
Benchmark Data Sets

Name	Organism	Genes (n)	Observations (m)
D1	*S. cerevisiae*	5,716	2,577
D2	*A. thaliana*	18,373	5,102
D3	*A. thaliana*	18,380	16,838

A. Data Sets

The goal in designing our framework is to enable parallelization of BN structure learning algorithms that are *constraint-based* and use MB discovery as an intermediary step. The framework is agnostic to the underlying application area. To demonstrate performance and scalability, we chose the construction of gene networks – a rich application area that has Big Data sets and the need for constructing large-scale networks. BN learning algorithms have been successfully used in recovering gene networks from gene expression data sets [3]. In this application, the genes are modeled as random variables which correspond to the nodes of a BN and the edges of the BN correspond to the biological interactions between the genes. In our experiments, we consider the genome-scale version of this problem, i.e., learning networks with tens of thousands of genes using thousands of gene expression studies. We used three real gene expression data sets of different sizes, summarized in Table I.

D1 is a data set generated from the organism *Saccharomyces cerevisiae*, a species of yeast involved in baking and brewing. Tchourine et al. [51] created this data set of 2,577 observations each for 5,716 genes by combining data from multiple RNA-seq expression studies. The data sets *D2* and *D3* contain expression profiles for *Arabidopsis thaliana*, a model organism in plant biology with more than 23,000 genes. These data sets are constructed by collecting over 18,000 microarray images from public databases (ArrayExpress and GEO), and pre-processing them using standard microarray data analysis workflows for quality control and normalization. In order to study process-specific phenomena, it is necessary for plant biologists to consult multiple gene networks generated from many process-specific data sets. *D2* is a subset of *D3*, manually curated by a domain specialist and includes only those microarray experiments that were designed to study the development process in *A. thaliana*. *D2* and *D3* contain 5,102 and 16,838 observations for 18,373 and 18,380 genes, respectively. We used the method recommended by Friedman et al. [2] for discretizing the data sets.

In order to study the scalability of our implementations on data sets with larger number of variables, we generated three simulated data sets with $n = 30,000$ and $m = 10,000$ using the *pcalg* [38] software as follows. First, we construct three random DAGs with 30,000 variables of increasing edge density by specifying edge addition probability of 5×10^{-5}, 1×10^{-4}, and 5×10^{-4}. Then, we use the dependency structure specified

TABLE II
COMPARISON OF THE TIME TAKEN BY *BNLEARN* AND OUR SEQUENTIAL IMPLEMENTATIONS IN CONSTRUCTING THE BNS FOR THE BENCHMARK DATA SETS, MEASURED IN SECONDS, AND THE CORRESPONDING SPEEDUP

Algorithm	Data set	Run-time (s)		Speedup
		bnlearn	*Ours*	
GS	*D1*	8 720.0	240.1	36.3
	D2	×	6 760.3	N/A
	D3	×	18 695.0	N/A
IAMB	*D1*	975.9	624.6	1.6
	D2	40 605.7	14 529.8	2.8
	D3	84 403.1	46 603.2	1.8
Inter-IAMB	*D1*	992.0	624.1	1.6
	D2	40 819.0	14 559.0	2.8
	D3	89 839.7	48 442.4	1.9

The symbol x indicates that the run did not finish in four days.

by the three DAGs to sample 10,000 observations for all the variables. Finally, we discretize the data sets as described above. We refer to the three simulated data sets so obtained as *S1*, *S2*, and *S3*, respectively. All the data sets are stored in plain text format on a GPFS storage which is accessible from all the nodes on the cluster. We used a significance threshold (α) of 0.05 for learning BNs in all our experiments.

B. *Comparison With* Bnlearn

We used *bnlearn* v4.5 with *R* v3.6.0 for the experiments reported in this section.

1) Sequential Comparison: We compare in Table II the run-time of *bnlearn* with that of our optimized sequential implementation, described in Section IV-A, for learning the network from the benchmark data sets using the three algorithms. The run-times for both *bnlearn* and our method are proportional to the size of the data sets, with *D1* taking the shortest time and *D3* taking the longest. We also observed that the implementation of *bnlearn* for the *GS* algorithm is almost an order of magnitude slower than that of the other two algorithms. This is because *bnlearn* implements the variable selection for the *GS* algorithm using expensive loops in *R*. Consequently, our implementation of the *GS* algorithm is 36.3X faster than *bnlearn* for learning the network for the *D1* data set. Further, *bnlearn* is not able to finish learning the network when using the *GS* algorithm for the two bigger data sets even after running for the cutoff time period of four days. For both the *IAMB* and the *Inter-IAMB* algorithms, our sequential implementation outperforms *bnlearn* with a speedup of 1.6–2.8X for the benchmark data sets. Note that our implementation of the *GS* algorithm is 2–3X faster than the other two algorithms because of the optimization discussed in Section IV-B2.

We validated the networks learned by our implementations against those learned by *bnlearn* for the data set *D1* using the three algorithms. During the validation process, we discovered a bug in the *Construct* $\mathcal{PC}$ *from* $\mathcal{MB}$ phase of the *bnlearn* implementation. It was caused by an erroneous assumption in the implementation that if there is only one element in the $\mathcal{MB}$ set of a variable then it must be in the $\mathcal{PC}$ set of that variable. This bug was acknowledged as such by the package's maintainer (personal communication, March 4, 2020). We fixed this bug in *bnlearn* and used the networks learned using this modified version for the purpose of the validation. For all the three algorithms, the networks learned using our implementations recall more than 99.84% of the edges present in the networks learned using the corresponding implementations from *bnlearn* with more than 99.92% precision, i.e., our implementations learn more than 99.84% of the edges in the networks learned by *bnlearn* with less than 0.08% additional edges.

2) Parallel Scalability of Bnlearn*:* As we discussed in Section I-A, most parallelization strategies for BN learning focus on either *score-based* or *global-search constraint-based* methods. Among the software for BN structure learning that we surveyed, *bnlearn* is the only one that supports learning BNs on multiple cores using the three algorithms that we focus on. It uses the *parallel* library of the core *R* distribution for parallelizing the structure learning using a master-worker paradigm on the specified cores [39]. Since *bnlearn* is the only other available parallel implementation of the three algorithms under consideration, we evaluate its parallel scalability as a baseline for our method.

We use the three algorithms implemented in *bnlearn* for learning the BNs from the benchmark data sets using an increasing number of cores and measure their self-speedup, i.e., speedup compared to the sequential run-time of the *bnlearn* implementation. When using 2 cores, the *IAMB* algorithm shows a speedup of 1.8X, 1.3X, and 1.6X for *D1*, *D2*, and *D3* respectively. *bnlearn* shows further improvement when using 16 cores with an observed speedup of 6.3X, 2.0X, and 3.4X. However, the speedup starts flattening when using cores on multiple nodes. For example, when using 64 cores on four nodes, the observed speedup is 7.7X, 2.1X, and 3.9X – a marginal improvement over the speedup using 16 cores. Speedup for the other two algorithms showed a similar pattern of diminishing returns. Since *bnlearn*'s parallel scalability deteriorates significantly with increasing number of cores, we do not explore its performance further here. The scalability of our implementations, presented in Section V-D, outperforms *bnlearn* by a significant margin.

C. *Effect of Load Balancing*

In order to understand the extent of load imbalance during the parallel execution of the three algorithms, we learned BNs from data set *D2* using the algorithms, without the application of load balancing strategies discussed in Section IV-C, and recorded the imbalance (as per (4)) at the end of each iteration. We observe that the imbalance during execution on less than 16 cores stays close to 1. However, the imbalance increases when the algorithms are executed on larger number of cores with more and more processes left without any work as the algorithms progress. Further, the imbalance increases with an increase in the number of iterations of the algorithm. For example, the *IAMB* algorithm runs for 5 iterations for the data set and shows a final imbalance of 15.6 when run on 512 cores while the *GS* and the *Inter-IAMB* algorithms show a final imbalance tending to ∞, both of which take 7 iterations for completion.

The percentage reduction in the run-time for learning the network from *D2* data set, with the application of the

TABLE III
TIME TAKEN IN LEARNING THE BNS FOR THE BENCHMARK DATA SETS USING THE THREE ALGORITHMS ON DIFFERENT NUMBER OF CORES, MEASURED IN SECONDS

Number of	*GS*			*IAMB*			*Inter-IAMB*		
Cores	*D1*	*D2*	*D3*	*D1*	*D2*	*D3*	*D1*	*D2*	*D3*
1	240.1	6 760.3	18 695.0	624.6	14 529.8	46 603.2	624.1	14 559.0	48 442.4
2	133.7	3 615.7	10 123.7	330.1	7 702.9	24 677.3	333.2	7 691.4	25 616.2
4	71.0	1 900.0	5 299.5	170.5	3 957.3	12 626.3	172.2	3 932.8	13 182.5
8	39.0	1 043.4	2 930.5	88.6	2 053.3	6 609.3	89.9	2 067.5	6 955.0
16	20.7	538.5	1 526.7	45.2	1 046.1	3 358.6	45.9	1 061.5	3 498.6
32	11.5	295.3	839.3	23.0	529.8	1 716.4	23.4	533.3	1 802.3
64	6.1	146.5	412.5	11.9	260.6	841.8	12.1	262.4	867.3
128	3.4	83.3	242.6	6.2	130.9	426.3	6.4	132.4	445.2
256	2.1	44.6	134.5	3.2	66.5	209.7	3.4	66.9	221.2
512	1.5	24.7	76.3	1.8	34.0	106.6	2.0	34.9	113.1
1024	1.5	14.1	43.1	1.4	17.7	55.2	2.3	18.8	59.1

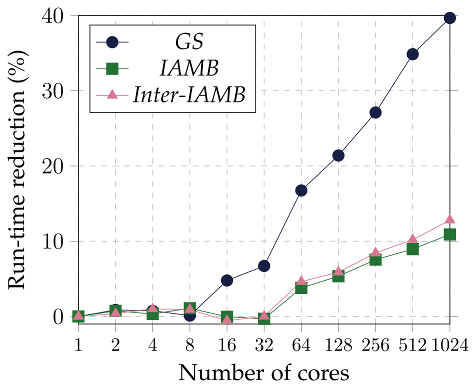

Fig. 2. Plot of percentage reduction in the run-time, as a result of load balancing, of the three algorithms used for learning BN from data set *D2* on different number of cores.

redistribution strategy and using the three algorithms for different number of cores, is shown in Fig. 2. When running on fewer cores, we observe almost no improvement in run-time with load balancing because the observed imbalance is small. Even when the imbalance is high, the time taken in measuring the imbalance and redistributing may be more than the corresponding gains. In such cases, we observe that the run-time increases marginally when load balancing is enabled, with the highest observed increase of just 0.6% when using the *Inter-IAMB* algorithm on 16 cores. However, when running on larger number of cores, all the algorithms show significant reduction in the run-times with the *GS* algorithm showing close to 40% improvement on 1,024 cores. Since the optimization for the *GS* algorithm, discussed in Section IV-B2, enables faster candidate selection for many variables, the algorithm benefits more from a better spread of the work load through an evenly distributed *c-scores* list. This is the reason we observe a significantly higher improvement in the case of the *GS* algorithm, as compared to the other two algorithms.

D. Scalability of Our Framework

Our procedure to read an input data set in parallel is as follows. First, the rows of the data set are block distributed to all the MPI processes. Then, the processes concurrently read the discretized data from their assigned rows. Finally, the read data is collected on all the processes to get the complete data set using `MPI_Allgatherv`. Once the BN is constructed, the corresponding network is written in *graphviz* [52] format. In our experiments, we observed that reading the data sets takes in the range of $0.6 - 3.7$ seconds for *D1*, $3.9 - 23.5$ seconds for *D2*, $12.0 - 77.3$ seconds for *D3*, and $9.4 - 76.2$ seconds for the simulated data sets. Writing out the learned network takes less than 0.3 seconds in all the cases. For scalability discussions, we report only the time taken for constructing the BN by the parallel algorithm implementation and not for the I/O.

1) Strong Scaling for Benchmark Data Sets: We conducted strong scaling experiments for all the three algorithms using the benchmark data sets by repeatedly doubling the number of cores from 1 to 1,024. Table III shows the average run-times for all the combinations of the algorithms, cores, and data sets. To better understand the performance of our implementations, we computed strong scaling speedup and efficiency as follows:

$$\text{Speedup} = \frac{T_1}{T_p} \quad \& \quad \text{Efficiency } (\%) = \frac{T_1}{p \cdot T_p} \times 100,$$

where p is the number of cores used, T_1 is the run-time of the best sequential algorithm, and T_p is the run-time when using p cores. The strong scaling speedup of the three algorithms for the benchmark data sets as the number of cores used is increased are plotted in the first row of Fig. 3 and the corresponding plots of efficiency are shown in the second row. Note that a perfect parallel implementation would achieve linear speedup and 100% efficiency.

As can be observed from the figure, our implementations of all the three algorithms show near-linear scaling on up to 1,024 cores for the two larger data sets (*D2* and *D3*), while the scaling tapers off on more than 256 cores for the smaller data set (*D1*). The poor scaling for *D1* on larger number of cores can be explained by the lower total work required for learning the BN from this data set, as demonstrated by the corresponding run-time of less than 3.4 seconds for all the algorithms on 256 cores and above. The *IAMB* and the *Inter-IAMB* algorithms achieve a strong scaling efficiency of more than 75% when run on up to 1,024 cores for data sets *D2* and *D3*.

The lower efficiency of *GS* is because the optimization discussed in Section IV-B2 reduces the total work required by

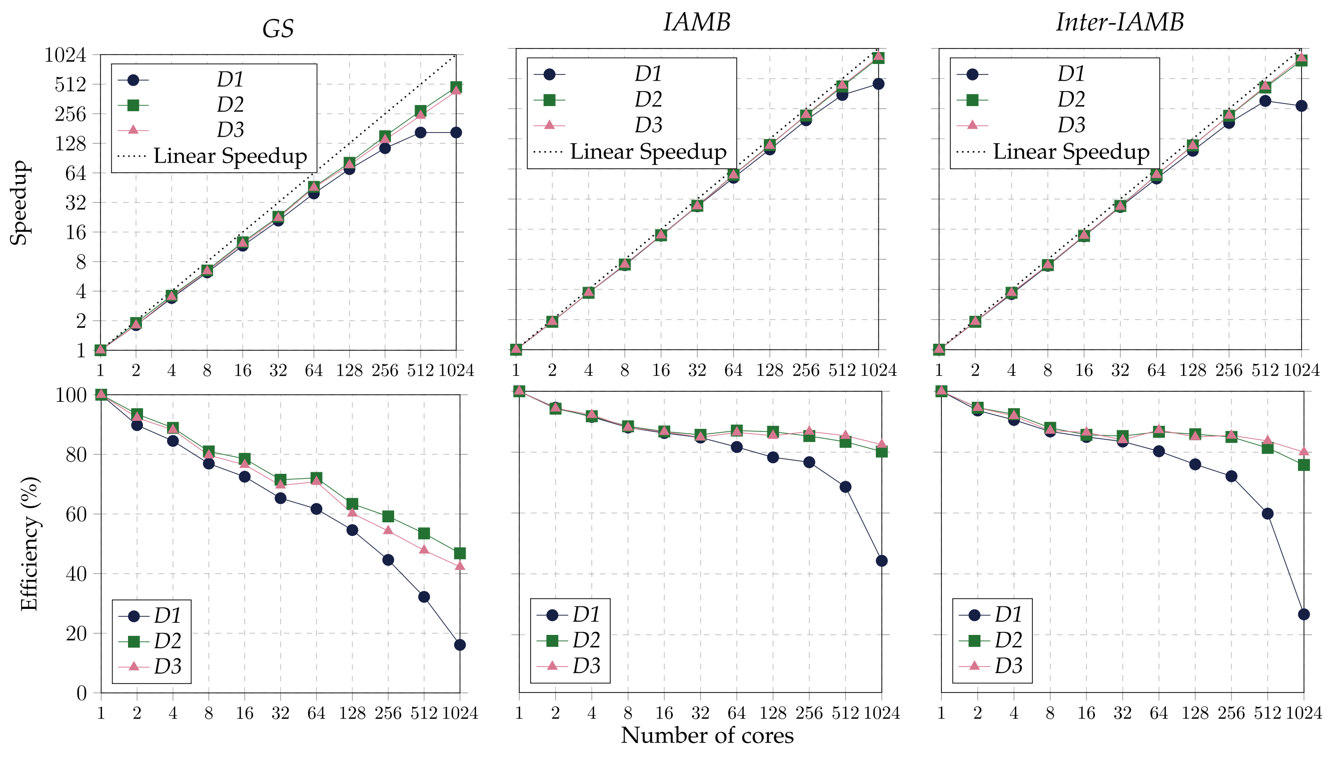

Fig. 3. Plots of strong scaling speedup and efficiency of the three algorithms in constructing the BNs for the benchmark data sets as a function of the number of cores.

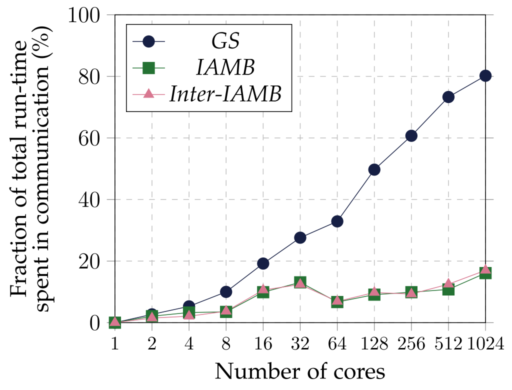

Fig. 4. Plot of fraction of total run-time spent in communication by the three algorithms used for learning the BN from *D2* data set.

the algorithm. On larger number of cores, this reduction in total work leads to lower computation load per processor, as compared to the other two algorithms, and therefore the run-time of the *GS* algorithm is dominated by communication time. For instance, the fraction of the total run-time spent by the three algorithms in communication while learning BNs from *D2* is shown in Fig. 4. These plots demonstrate a markedly higher communication overhead for the *GS* algorithm when running on larger number of cores. However, despite the lower efficiency, the optimization helps the *GS* algorithm achieve a speedup of up to 1.6X over the *IAMB* and the *Inter-IAMB* on 1024 cores.

Our implementations of the three algorithms are able to learn BNs from the benchmark data sets in less than a minute on 1,024 cores, with a maximum speedup of 844.8X and a corresponding 82.5% scaling efficiency. To demonstrate that our algorithms scale to well beyond 1024 cores, we ran them using 1408 cores with 22 cores per node over the 64 nodes available. With 1,408 cores, the run-times of the *GS*, the *IAMB* and the *Inter-IAMB* algorithms for data set *D3* are 38.4, 45.5, and 47.0 seconds, respectively. Even with an increase in the number of cores used per node, these run-times correspond to an additional speedup of 54X, 180X, and 212X compared to the speedup obtained using 1,024 cores while incurring less than 10% loss in strong scaling efficiency.

2) Strong Scaling for Simulated Data Sets: For learning BNs from the simulated data sets with even larger number of variables, all the three algorithms show near-linear scaling. In particular, scalability of the *GS* algorithm improves significantly compared to the benchmark data sets. Our optimized sequential implementation of *GS* learns the network for *S1*, *S2*, and *S3* in 9.7, 13.0, and 20.8 hours, respectively. Using our parallel implementation on 1024 cores, the corresponding run-times are 52.0, 74.0, and 105.6 seconds. Strong scaling efficiency of the the *GS* algorithm for the simulated data sets is plotted in Fig. 5. The considerable increase in efficiency when compared to what is observed for the benchmark data sets (Fig. 3) is in line with our discussion on the efficiency of the algorithm in the previous section. Since there is more total work required for learning networks from the simulated data sets, the computation load per processor of the algorithm is high even when running on 1,024 cores. Correspondingly, the fraction of run-time spent by the algorithm in communication for these data sets on 1,024 cores is between 38.3% and 44.3%, which is almost half of that observed for the benchmark data sets. Further, the average

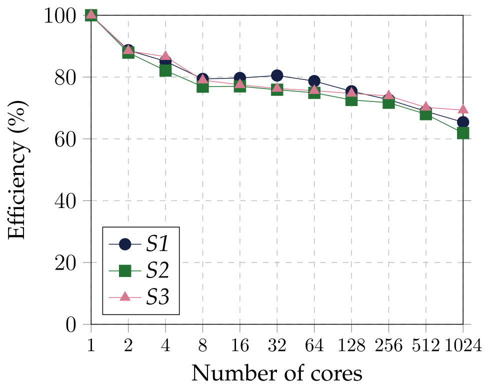

Fig. 5. Plot of strong scaling efficiency of the *GS* algorithm in constructing the BNs for the simulated data sets.

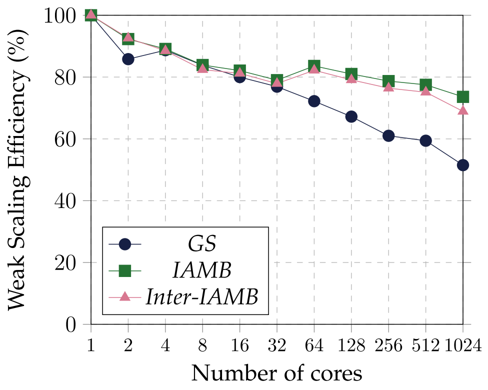

Fig. 6. Plot of weak scaling efficiency of the three algorithms, measured for data set *D2*.

size of MBs learned by the *GS* algorithm, as measured after the *Grow* phase, increases from 3.83 for *S1* to 6.07 for *S3*. Therefore, fewer variable pairs are eliminated from *c-scores* list during the execution of the algorithm for *S3* as compared to *S1*, resulting in higher total work which enables better scaling efficiency of the algorithm for *S3* on 1,024 cores.

Our parallel implementations are able to reduce the time required for learning BNs from more than a day for a sequential run to less than two minutes using 1,024 cores. The sequential run-times of the *IAMB* algorithm for *S1*, *S2*, and *S3* are 13.4, 16.4, and 24.7 hours and the run-times on 1,024 cores are 58.5, 73.5, and 105.3 seconds, respectively. Similarly, the sequential run-time of the *Inter-IAMB* algorithm for the three data sets are 13.8, 16.9, and 25.6 hours while the run-times on 1024 cores are 59.7, 75.7, and 120.0 seconds. Both *IAMB* and *Inter-IAMB* show more than 75% strong scaling efficiency on 1,024 cores for learning BNs and a maximum observed speedup of 845X corresponding to scaling efficiency of 82.5%.

3) Weak Scaling: We performed weak scaling experiments with data set *D2* using the same set of cores as the strong scaling experiments. For the runs using 1,024 cores, we use the complete *D2* data set. When using p cores (where $p < 1024$), we learn the BN for a subset of n_p variables from the complete data set (where $n_p < n$) such that n_p^2/p remains the same (to keep the work load per core approximately the same regardless of the number of cores used). We compute the weak scaling efficiency as $\frac{T_1}{T_p} \times 100$, where T_1 is the run-time of the best sequential algorithm for learning the BN from n_1 variables and T_p is the run-time of the parallel implementation in learning the BN from n_p variables using p cores. The plots of weak scaling efficiency of the three algorithms are shown in Fig. 6. The degradation in scaling efficiency for the three different algorithms is in line with the communication intensity of the respective algorithms (Fig. 4), which suggests that communication overhead is the limiting factor for weak scaling.

VI. Reproducibility

Reproducibility of results is vital in any scientific field that relies on experiments. With the aim of promoting reproducibility in the field of high-performance computing, the International Conference for High Performance Computing, Networking, Storage and Analysis, also known more widely as the Supercomputing Conference Series (SC) has included a reproducibility challenge as part of the student cluster competition (SCC) since 2016 [53]. The challenge tasks teams of undergraduate students with reproducing the results of a paper from the previous year's SC conference. We are honored that, based on its Artifact Descriptor (AD) and its suitability to the SCC, our paper from SC 2020 [54] was selected for the reproducibility challenge component of the 2021 SCC (SCC21) [55].

Ten teams, each consisting of six undergraduate students, an advisor, and vendor partners, were selected to participate in SCC21. These teams utilized infrastructure provided by the Oracle Cloud for the reproducibility challenge in SCC21. We provide more details about the infrastructure used by the teams in Section VI-A. In order to keep the competition unbiased, new data sets were created and used for the challenge, as described in Section VI-B. The teams were tasked with using the challenge data sets to reproduce the results from the original paper. The critiques of the four top performing teams on the reproducibility challenge have been included as part of this special edition. The top teams, in alphabetical order, are from Peking University [56], ShanghaiTech University [57], Tsinghua University [58], and UC San Diego [59]. We discuss the results presented in these critiques in Section VI-C.

A. Infrastructure

1) Hardware Resources: Traditionally, SCC teams are expected to build their own clusters during the competition and then complete the reproducibility challenge on these custom-built clusters. However, during 2020 and 2021, the competition was forced to be completely virtual due to the COVID-19 pandemic. As a result, the reproducibility challenge was conducted using cloud resources for these two years. During SCC21, teams were provided access to the infrastructure sponsored by the Oracle Cloud for the reproducibility challenge.

In the interest of fairness, all the teams had access to an identical cloud compute cluster. Each cluster consisted of an 8-core *VM.Optimized3.Flex* login node and four 36-core

TABLE IV
COMPARISON OF THE HARDWARE RESOURCES USED FOR THE ORIGINAL EXPERIMENTS WITH THAT USED FOR THE REPRODUCIBILITY EXPERIMENTS

	Original Experiments	Reproducibility Experiments
Computation Resources		
Processor Name	Intel Xeon 6226	Intel Xeon 6354
Processor Speed	2.7 GHz	3.0 GHz
Number of Cores	24	36
Memory Per Node	$\geq$ 192 GB	512 GB
Number of Nodes	64	4
Total Number of Cores	64×24	4×36
Network Resources		
Interconnect	EDR InfiniBand	RoCE v2
Network Bandwidth	$\leq$ 100 Gbps	$\leq$ 100 Gbps

BM.Optimized3.36 compute nodes, where all the nodes had a 3.0 GHz Intel Xeon 6354 processor and the compute nodes had 512 GB of main memory. The clusters also had access to GPU nodes that were not used for the reproducibility challenge. The compute nodes were connected by Remote Direct Memory Access over Converged Ethernet (RoCE) v2 with a maximum network bandwidth of 100 Gbps. Each cluster also had 1 TB of NFS storage that was accessible from all the nodes in the cluster. The differences between the hardware resources used for the original experiments and those used for the reproducibility experiments are summarized in Table IV.

2) Software Setup: All the CPU nodes on the cloud ran *Oracle Linux 7.9*. The teams were expected to install other software and libraries as per their requirements. While the team from Peking University built the required libraries from source, the other teams utilized package managers, such as *spack* (used by the teams from ShanghaiTech University and UC San Diego) and *nix* (used by the Tsinghua University team), for the purpose. All these teams used different versions of *gcc* as the C++ compiler, in combination with three different MPI implementations: the teams from Peking University and UC San Diego used *MVAPICH2*, the team from Tsinghua University used *OpenMPI*, and *HPC-X* was used by the team from ShanghaiTech University. There were also minor differences in the version of Boost C++ Library [60], as well as that of the build tool – SCons [61]. The differences in the software setup used for the original experiments with that used by the four teams are summarized in Table V.

Details about the setup of the four teams can be found in the corresponding critiques, where the teams have also documented their efforts to compile the application for optimal performance on the cloud. For example, the team from Tsinghua University reported trying out different MPI implementations and choosing the one with the lowest bandwidth and latency on OSU benchmarks, while the team from ShanghaiTech University reported experimenting with different MPI compiler flags. Some teams also reported the challenges that they faced during this process, e.g., the team from UC San Diego reported having to install a specific version of Slurm to work with MVAPICH, and the team from ShanghaiTech University reported that the application failed to compile with the Intel C++ compiler. In spite of these challenges, all the top teams were able to build versions of applications that seemed to show similar scalability as the one used for the original experiments.

B. Challenge Data Sets

In collaboration with the SCC21 reproducibility challenge committee, we chose the same application area for the challenge as the one described in Section V-A – construction of gene networks. The importance of this application area has grown even more during the COVID-19 pandemic. This is because understanding the widely varying genetic response to the novel coronavirus SARS-CoV-2 in humans is critical to developing vaccines and treatments for the disease. Further, multiple recent studies have created COVID-19 related data sets and made them available online. One such study was done by Ziegler et al. [62] that obtained upper respiratory tract swabs from 58 individuals. Then, they sequenced all the cells recovered from the swabs, using single-cell RNA-sequencing, to get 32,871 gene expression values from 32,588 cells. We used the data set from [62] to generate three different data sets for the challenge as described below.

First, we divided all the cells into three categories, using the World Health Organization (WHO) categorization of individuals that they were obtained from [63]: 23 individuals who had tested negative for COVID-19, 14 individuals who had tested positive and showed mild symptoms (WHO classification of $1 - 5$), and 21 individuals who had tested positive and exhibited severe symptoms (WHO classification of $6 - 8$). Then, we further refined the three data sets so obtained by only retaining genes and cells with at least one non-zero value to get the final data sets that were used for the challenge. These data sets are described in Table VI.

The smallest data set, *C1*, was provided to the teams two weeks ahead of the competition for testing purposes. The two bigger data sets, *C2* and *C3*, were used as the challenge data sets and were made available to the teams at the beginning of the competition. Notice that, both *C2* and *C3* are bigger than all the data sets that we used to obtain the results presented in Section V, including the bigger simulated data sets (*S1*, *S2*, and *S3*).

C. Experiments and Results

The problem statement, defined by the reproducibility challenge committee, asked the SCC21 teams to use data sets *C2* and *C3* to reproduce the following results presented in Section V:

1) Perform a strong scaling study (similar to Section V-D1).
   - Create a table with the strong scaling run-times of the three algorithms (similar to Table III).
   - Generate strong scaling plots for the three algorithms (similar to Fig. 3). Also generate a communication overhead plot for the three algorithms using data set *C2* (similar to Fig. 4).
2) Perform a weak scaling study (similar to Section V-D3).
   - Generate weak scaling efficiency plots for the three algorithms using data set *C2* (similar to Fig. 6).

The teams had access to the cloud compute clusters for a 48-hour period during SCC21 to conduct these experiments.

TABLE V
Comparison of the Software Setup Used for the Original Experiments With That Used for the Reproducibility Experiments by the Four Teams

	Original Experiments	Reproducibility Experiments			
		Peking University	ShanghaiTech University	Tsinghua University	UC San Diego
Operating System	*RHEL 7.6*	*Oracle Linux 7.9*			
Package Manager	*spack*	None	*spack*	*nix*	*spack*
C++ Compiler	*gcc v9.2.0*	*gcc v9.3.0*	*gcc v9.2.0*	*gcc v10.3.0*	*gcc v9.2.0*
MPI Implementation	*MVAPICH v2.3.3*	*MVAPICH v2.3.3*	*HPC-X v2.8.1*	*OpenMPI v4.1.1*	*MVAPICH v2.3.6*
Boost Library	*v1.70.0*	*v1.70.0*	*v1.70.0*	*v1.69.0*	*v1.77.0*
SCons	*v3.1.2*	*v4.2.0*	Unknown	*v4.1.0*	*v4.2.0*

TABLE VI
Reproducibility Challenge Data Sets

Name	COVID-19 Test	Symptoms	Genes (n)	Cells (m)
C1	*+ve*	*mild*	29,150	5,164
C2	*-ve*	*none*	30,307	11,180
C3	*+ve*	*severe*	30,604	12,909

Due to the limited time they had to run all the experiments, the teams came up with different strategies to ensure that they were able to finish them in time. This included running multiple experiments in parallel to maximally utilize the resources that were allocated to each team. These strategies enabled three out of the four teams to obtain results from at least one run for all the experiments. The only exception was the team from UC San Diego that encountered OS/Slurm errors for a few runs on 64 and 128 cores [59]. They attributed these errors to memory constraints of the cluster. However, since the other three teams used exactly the same hardware resources and did not observe this behavior, we believe that these errors may have been caused by the differences in the software setup used, potential interference by other jobs running in parallel, or transient issues with the resources allocated to the team. We discuss the results of the scaling studies done by the four teams in detail below.

1) Strong Scaling Results: For the strong scaling experiments, the top four teams selected the number of cores using an approach similar to the one used for the results in Section V. We had varied the number of cores used between 1 and 1,024, always using power of two number of cores for our experiments. Correspondingly, since the teams had access to $4 \times 36 = 144$ cores, they varied the number of cores used between 1 and 128 for their results.

The reported strong scaling behavior of the three algorithms for learning BNs from the two challenge data sets was consistent across the four teams. Fig. 7 shows the strong scaling efficiency observed by three of the four teams that reported run-times in their critiques, in constructing the BNs from the two data sets. All the teams observed greater than 70% strong scaling efficiency when using *IAMB* and *Inter-IAMB*, and greater than 40% efficiency when using *GS* on up to 128 cores. They also observed lower communication overhead for *IAMB* and *Inter-IAMB* (less than 20% when using any number of cores) as compared to that of *GS* (more than 60% on 128 cores). Intriguingly, even though the teams performed the scaling experiments using the bigger challenge data sets, the strong scaling efficiency reported by the teams was about the same or lower than the corresponding efficiency that we observed on 128 cores using benchmark data sets for all the algorithms. The reported lower efficiency is in agreement with the higher communication overhead observed by the teams, which may be attributed to the higher latency of Ethernet switches used by RoCE interconnect as compared to the lower latency dedicated Infiniband switches used by the interconnect on Hive cluster.

Despite the lower efficiency observed by the SCC teams, the strong scaling results of their experiments follow the general trend that we observed when learning from the two bigger benchmark data sets (*D2* and *D3*) in Section V-D1. Similar to the strong scaling speedup and efficiency shown in Fig. 3, all the teams observed near-linear speedup when learning from *C2* and *C3* using *IAMB* and *Inter-IAMB*, but lower speedup when using *GS*. Further, all the teams observed a communication overhead plot for *C2* that was similar to Fig. 4, and thus attributed the lower efficiency of *GS* to its higher communication overhead.

2) Weak Scaling Results: The four teams also conducted the weak scaling experiments by varying the number of cores between 1 and 128, similar to the strong scaling experiments. To compute the number of variables on different number of cores, the teams followed the same methodology that we used for the purpose and described in Section V-D3. While the teams from ShanghaiTech University, Peking University, and Tsinghua University selected the first n_p variables to construct the data set for learning on $p < 128$ cores, the team from UC San Diego used random sampling for the purpose [59]. Consequently, the former three teams observed very similar weak scaling behavior with weak scaling efficiency of greater than 60% for *IAMB* and *Inter-IAMB*, and greater than 40% for *GS* on all cores. On the other hand, the UC San Diego team reported higher scaling efficiency on all the cores: more than 80% for *IAMB* and *Inter-IAMB*, and greater than 60% for *GS*. Interestingly, they also reported the lowest efficiency for *GS* on 16 cores and for *IAMB* and *Inter-IAMB* on 8 cores.

Irrespective of the aforementioned differences between the results obtained by the four teams, the weak scaling efficiency plots of all the teams correspond well to Fig. 6. In general, the teams observed lower efficiency when using larger number of cores. Further, the efficiency when using *GS* was lower as compared to *IAMB* and *Inter-IAMB*. This further substantiates our

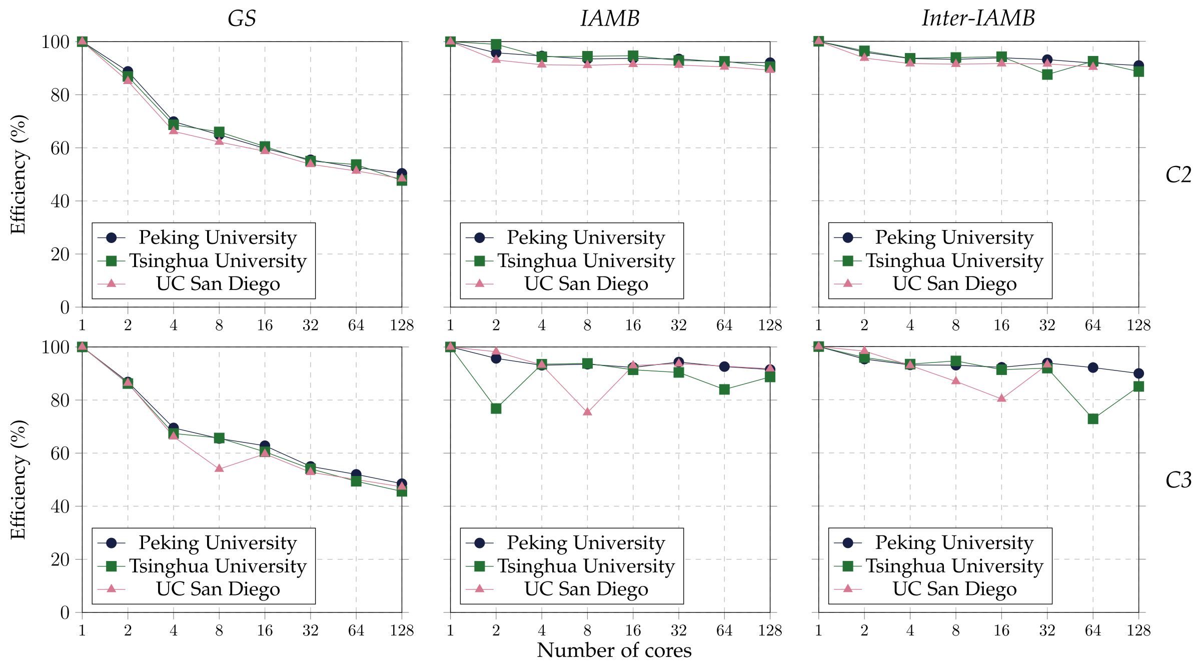

Fig. 7. Plots of strong scaling efficiency of the three algorithms reported by three SCC21 teams in constructing the BNs for the reproducibility challenge data sets *C2* (top row) and *C3* (bottom row).

conclusion in Section V-D3, that the communication overhead of the algorithms dictates their weak scaling efficiency.

VII. Conclusion and Future Work

We presented a framework for parallelizing multiple BN structure learning algorithms. The algorithms implemented using this framework, as part of an open-source software called *ramBLe*, are able to construct genome-scale networks for two model organisms, *S. cerevisiae* and *A. thaliana*, in less than a minute on 1,024 cores. The scalability of our implementations of the algorithms was independently verified by the SCC21 teams for construction of gene networks from even larger COVID-19 data sets.

Our implementations can greatly aid biologists in their research on gene networks because it can save weeks of their time during the iterative search for the optimal parameters to construct a BN that best approximates the biological truth. Moreover, our parallel implementations show good scaling for learning networks with larger number of variables from simulated data sets and can therefore be used for other applications which require learning high dimensional causal networks.

Directions for future research include efficiently conducting CI tests when data sets are distributed across all processors, balancing workloads arising from different methodologies for CI tests (e.g., permutation tests, bootstrapping, etc.) for discrete as well as continuous data, and extending the framework to include other categories of BN learning algorithms.

Acknowledgments

We thank Maneesha Aluru for providing customized *Arabidopsis thaliana* data sets used in this work, and Tony Pan for guidance on debugging MPI performance. We also thank Le Mai Weakley and the other SCC21 reproducibility challenge committee members for their help and guidance with preparing the application for the challenge, and Junjie Li for his assistance with replicating the original results and testing the challenge data sets using different computation resources.

References

[1] E. Kyrimi, S. McLachlan, K. Dube, M. R. Neves, A. Fahmi, and N. Fenton, “A comprehensive scoping review of Bayesian networks in healthcare: Past, present and future,” 2020, *arXiv:2002.08627*.
[2] N. Friedman, M. Linial, I. Nachman, and D. Pe’er, “Using Bayesian networks to analyze expression data,” *J. Comput. Biol.*, vol. 7, no. 3/4, pp. 601–620, 2000.
[3] S. Imoto, T. Higuchi, T. Goto, K. Tashiro, S. Kuhara, and S. Miyano, “Combining microarrays and biological knowledge for estimating gene networks via Bayesian networks,” *J. Bioinf. Comput. Biol.*, vol. 2, no. 01, pp. 77–98, 2004.
[4] J. Ramsey, M. Glymour, R. Sanchez-Romero, and C. Glymour, “A million variables and more: The fast greedy equivalence search algorithm for learning high-dimensional graphical causal models, with an application to functional magnetic resonance images,” *Int. J. Data Sci. Analytics*, vol. 3, no. 2, pp. 121–129, 2017.
[5] X. Sun, J. Dai, P. Liu, A. Singhal, and J. Yen, “Using Bayesian networks for probabilistic identification of zero-day attack paths,” *IEEE Trans. Inf. Forensics Secur.*, vol. 13, no. 10, pp. 2506–2521, Oct. 2018.
[6] C. S. Vlek, H. Prakken, S. Renooij, and B. Verheij, “A method for explaining Bayesian networks for legal evidence with scenarios,” *Artif. Intell. Law*, vol. 24, no. 3, pp. 285–324, 2016.
[7] F. Taroni, A. Biedermann, S. Bozza, P. Garbolino, and C. Aitken, *Bayesian Networks for Probabilistic Inference and Decision Analysis in Forensic Science*. Hoboken, NJ, USA: Wiley, 2014.
[8] D. Gunning and D. W. Aha, “Darpa’s explainable artificial intelligence program,” *AI Mag.*, vol. 40, no. 2, pp. 44–58, 2019.
[9] C. Yuan, H. Lim, and T.-C. Lu, “Most relevant explanation in Bayesian networks,” *J. Artif. Intell. Res.*, vol. 42, pp. 309–352, 2011.
[10] C. Rudin, “Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead,” *Nature Mach. Intell.*, vol. 1, no. 5, pp. 206–215, 2019.

[11] D. M. Chickering, D. Heckerman, and C. Meek, “Large-sample learning of Bayesian networks is np-hard,” *J. Mach. Learn. Res.*, vol. 5, no. Oct, pp. 1287–1330, 2004.

[12] M. Koivisto and K. Sood, “Exact Bayesian structure discovery in Bayesian networks,” *J. Mach. Learn. Res.*, vol. 5, no. May, pp. 549–573, 2004.

[13] T. Silander and P. Myllymäki, “A simple approach for finding the globally optimal Bayesian network structure,” in *Proc. 22nd Conf. Uncertainty Artif. Intell.*, Arlington, Virginia, USA: AUAI Press, 2006, pp. 445–452.

[14] Y. Tamada, S. Imoto, and S. Miyano, “Parallel algorithm for learning optimal Bayesian network structure,” *J. Mach. Learn. Res.*, vol. 12, no. Jul, pp. 2437–2459, 2011.

[15] O. Nikolova, J. Zola, and S. Aluru, “Parallel globally optimal structure learning of Bayesian networks,” *J. Parallel Distrib. Comput.*, vol. 73, no. 8, pp. 1039–1048, 2013.

[16] Y. Tamada, “Memory efficient parallel algorithm for optimal dag structure search using direct communication,” *J. Parallel Distrib. Comput.*, vol. 119, pp. 27–35, 2018.

[17] G. F. Cooper and E. Herskovits, “A Bayesian method for the induction of probabilistic networks from data,” *Mach. Learn.*, vol. 9, no. 4, pp. 309–347, 1992.

[18] D. Heckerman, D. Geiger, and D. M. Chickering, “Learning Bayesian networks: The combination of knowledge and statistical data,” *Mach. Learn.*, vol. 20, no. 3, pp. 197–243, 1995.

[19] P. Spirtes, C. Glymour, and R. Scheines, “An algorithm for fast recovery of sparse causal graphs,” *Social Sci. Comput. Rev.*, vol. 9, no. 1, pp. 62–72, 1991.

[20] P. Spirtes and C. Meek, “Learning Bayesian networks with discrete variables from data,” in *Proc. 1st Int. Conf. Knowl. Discov. Data Mining*, AAAI Press, 1995, vol. 1, pp. 294–299.

[21] T. Niinimäki and P. Parviainen, “Local structure discovery in Bayesian networks,” in *Proc. 28th Conf. Uncertainty Artif. Intell.*, AUAI Press, 2012, pp. 634–643.

[22] T. Gao, K. Fadnis, and M. Campbell, “Local-to-global Bayesian network structure learning,” in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 1193–1202.

[23] D. Margaritis and S. Thrun, “Bayesian network induction via local neighborhoods,” in *Proc. Adv. Neural Inf. Process. Syst.*, MIT Press, 2000, pp. 505–511.

[24] I. Tsamardinos, C. F. Aliferis, A. R. Statnikov, and E. Statnikov, “Algorithms for large scale Markov blanket discovery,” in *Proc. FLAIRS Conf.*, AAAI Press, 2003, vol. 2, pp. 376–380.

[25] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. Cambridge, MA, USA: MIT Press, 2009.

[26] O. Nikolova and S. Aluru, “Parallel Bayesian network structure learning with application to gene networks,” in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 1–9.

[27] S. Misra et al., “Parallel Bayesian network structure learning for genome-scale gene networks,” in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 461–472.

[28] A. L. Madsen, F. Jensen, A. Salmerón, H. Langseth, and T. D. Nielsen, “A parallel algorithm for Bayesian network structure learning from large data sets,” *Knowl.-Based Syst.*, vol. 117, pp. 46–55, 2017.

[29] D. Colombo and M. H. Maathuis, “Order-independent constraint-based causal structure learning,” *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 3741–3782, 2014.

[30] T. Le, T. Hoang, J. Li, L. Liu, H. Liu, and S. Hu, “A fast PC algorithm for high dimensional causal discovery with multi-core PCs,” *IEEE/ACM Trans. Comput. Biol. Bioinf.*, vol. 16, no. 5, pp. 1483–1495, Sep./Oct. 2019.

[31] C. Schmidt, J. Huegle, and M. Uflacker, “Order-independent constraint-based causal structure learning for gaussian distribution models using GPUs,” in *Proc. 30th ACM Int. Conf. Sci. Statist. Database Manage.*, 2018, pp. 1–10.

[32] B. Zarebavani, F. Jafarinejad, M. Hashemi, and S. Salehkaleybar, “cuPC: CUDA-based parallel PC algorithm for causal structure learning on GPU,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 3, pp. 530–542, Mar. 2020.

[33] O. Nikolova and S. Aluru, “Parallel discovery of direct causal relations and Markov boundaries with applications to gene networks,” in *Proc. IEEE Int. Conf. Parallel Process.*, 2011, pp. 512–521.

[34] I. Tsamardinos, L. E. Brown, and C. F. Aliferis, “The max-min hill-climbing Bayesian network structure learning algorithm,” *Mach. Learn.*, vol. 65, no. 1, pp. 31–78, 2006.

[35] J. M. Peña, R. Nilsson, J. Björkegren, and J. Tegnér, “Towards scalable and data efficient learning of Markov boundaries,” *Int. J. Approx. Reasoning*, vol. 45, no. 2, pp. 211–232, 2007.

[36] M. Scutari, “Learning Bayesian networks with the bnlearn R package,” *J. Statist. Softw.*, vol. 35, no. i03, pp. 1–22, 2010.

[37] R. Scheines, P. Spirtes, C. Glymour, C. Meek, and T. Richardson, “The tetrad project: Constraint based aids to causal model specification,” *Multivariate Behav. Res.*, vol. 33, no. 1, pp. 65–117, 1998.

[38] M. Kalisch, M. Mächler, D. Colombo, M. H. Maathuis, and P. Bühlmann, “Causal inference using graphical models with the R package pcalg,” *J. Statist. Softw.*, vol. 47, no. 11, pp. 1–26, 2012.

[39] M. Scutari, “Bayesian network constraint-based structure learning algorithms: Parallel and optimized implementations in the bnlearn R package,” *J. Statist. Softw.*, vol. 77, no. i02, pp. 1–20 2017.

[40] R. E. Neapolitan, *Learning Bayesian Networks*, vol. 38. Upper Saddle River, NJ, USA: Pearson Prentice Hall, 2004.

[41] P. Spirtes, C. N. Glymour, R. Scheines, and D. Heckerman, *Causation, Prediction, and Search*. Cambridge, MA, USA: MIT Press, 2000.

[42] C. F. Aliferis, I. Tsamardinos, and A. Statnikov, “HITON: A novel Markov blanket algorithm for optimal variable selection,” in *Proc. Annu. Symp. Amer. Med. Informat. Assoc.*, 2003, pp. 21–25.

[43] A. Srivastava, “ramBLe - A parallel framework for Bayesian learning,” 2020. [Online]. Available: https://github.com/asrivast28/ramBLe

[44] N. Noyes, K.-C. Cho, J. Ravel, L. J. Forney, and Z. Abdo, “Associations between sexual habits, menstrual hygiene practices, demographics and the vaginal microbiome as revealed by Bayesian network analysis,” *PLoS One*, vol. 13, no. 1, pp. 1–25, 2018.

[45] D. V. Zhernakova et al., “Individual variations in cardiovascular-disease-related protein levels are driven by genetics and gut microbiome,” *Nature Genet.*, vol. 50, no. 11, pp. 1524–1532, 2018.

[46] O. Delaneau et al., “Chromatin three-dimensional interactions mediate genetic effects on gene expression,” *Science*, vol. 364, no. 6439, 2019, Art. no. eaat8266.

[47] Z. Zheng et al., “Shared genetic control of root system architecture between zea mays and sorghum bicolor,” *Plant Physiol.*, vol. 182, no. 2, pp. 977–991, 2020.

[48] A. Cano, M. Gómez-Olmedo, and S. Moral, “A score based ranking of the edges for the PC algorithm,” in *Proc. 4th Eur. Workshop Probabilistic Graphical Models*, 2008, pp. 41–48.

[49] S. Karan, M. Eichhorn, B. Hurlburt, G. Iraci, and J. Zola, “Fast counting in machine learning applications,” in *Proc. 34th Conf. Uncertainty Artif. Intell.*, AUAI Press, 2018, pp. 540–549.

[50] B. S. Anderson and A. W. Moore, “ADtrees for fast counting and for fast learning of association rules,” in *Proc. 4th Int. Conf. Knowl. Discov. Data Mining*, AAAI Press, 1998, pp. 134–138.

[51] K. Tchourine, C. Vogel, and R. Bonneau, “Condition-specific modeling of biophysical parameters advances inference of regulatory networks,” *Cell Rep.*, vol. 23, no. 2, pp. 376–388, 2018.

[52] E. R. Gansner and S. C. North, “An open graph visualization system and its applications to software engineering,” *Soft. Pract. Exp.*, vol. 30, no. 11, pp. 1203–1233, 2000.

[53] M. A. Heroux and C. K. Garrett, “Special issue on SC16 student cluster competition reproducibility initiative,” *Parallel Comput.*, vol. 70, pp. 3–4, 2017.

[54] A. Srivastava, S. P. Chockalingam, and S. Aluru, “A parallel framework for constraint-based Bayesian network learning via Markov blanket discovery,” in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, pp. 1–15.

[55] L. M. Weakley, “SC21 student cluster reproducibility challenge committee converges on a benchmark,” 2021. [Online]. Available: https://sc21.supercomputing.org/2021/05/24/sc21-student-cluster-reproducibility-challenge-committee-converges-on-a-benchmark/

[56] J. Si et al., “Critique of “A parallel framework for constraint-based Bayesian network learning via Markov blanket discovery” by SCC team from Peking University,” *IEEE Trans. Parallel Distrib. Syst.*, to be published, doi: 10.1109/TPDS.2022.3206099.

[57] G. Li et al., “Critique of “A parallel framework for constraint-based Bayesian network learning via Markov blanket discovery” by SCC team from ShanghaiTech University,” *IEEE Trans. Parallel Distrib. Syst.*, to be published, doi: 10.1109/TPDS.2022.3205479.

[58] J. Cao et al., “Critique of “A parallel framework for constraint-based Bayesian network learning via Markov blanket discovery” by SCC team from Tsinghua University,” *IEEE Trans. Parallel Distrib. Syst.*, to be published, doi: 10.1109/TPDS.2022.3209723.

[59] A. Gupta et al., “Critique of: “A parallel framework for constraint-based Bayesian network learning via Markov blanket discovery” by SCC team from UC San Diego,” *IEEE Trans. Parallel Distrib. Syst.*, to be published, doi: 10.1109/TPDS.2022.3217284.

[60] B. Schäling, *The Boost C Libraries*. Boris Schäling, XML Press, 2011.
[61] S. Knight, "Building software with scons," *Comput. Sci. Eng.*, vol. 7, no. 1, pp. 79–88, 2005.
[62] C. G. Ziegler et al., "Impaired local intrinsic immunity to SARS-CoV-2 infection in severe COVID-19," *Cell*, vol. 184, no. 18, pp. 4713–4733, 2021.
[63] World Health Organization, "WHO R&D blueprint: Novel coronavirus COVID-19 therapeutic trial synopsis," 2020. [Online]. Available: https://www.who.int/publications/i/item/covid-19-therapeutic-trial-synopsis

Ankit Srivastava received the bachelor's degree from the Indian Institute of Technology, Kanpur, and the PhD degree from the School of Computational Science and Engineering within the College of Computing, Georgia Institute of Technology. He is a data scientist with Microsoft. Before this, he worked as part of the parallel computational fluid dynamics team of ANSYS, Inc. His research interests lie in the fields of high-performance computing, parallel algorithms, and Bayesian networks.

Sriram P. Chockalingam is a research scientist with the Institute for Data Engineering and Science (IDEaS), Georgia Institute of Technology. He develops high performance computing algorithms and implementations for IDEaS research efforts and collaborations. His research interests focus on development of sequential and parallel algorithms for network reverse engineering in systems biology, Bayesian network structure learning and approximate sequence matching with applications in Bioinformatics. He has more than a decade of experience in developing software in both industry and academia targeted towards solving data science problems.

Srinivas Aluru (Fellow, IEEE) is executive director of the Institute for Data Engineering and Science (IDEaS) and professor with the School of Computational Science and Engineering, Georgia Institute of Technology. He co-leads the NSF South Big Data Regional Innovation Hub which nurtures Big Data partnerships between organizations in the 16 Southern States and Washington D.C., and the NSF Transdisciplinary Research Institute for Advancing Data Science. He conducts research in high performance computing, large-scale data analysis, bioinformatics and systems biology, combinatorial scientific computing, and applied algorithms. He contributed to NSF and DOE led efforts for strategic development of Big Data and exascale computing. He is a fellow of AAAS, ACM, and SIAM, and is a recipient of the IEEE Computer Society Golden Core and Meritorious Service awards.